SproutCore Render Delegates Specification

Version 1.3
December 16, 2010

Authors:

Tom Dale tdale@apple.com

Alex Iskander aiskander@apple.com

1.3 Changes:

- Removed getDisplayProperties/getChangedDisplayProperties
- Added description of SC.View's renderDelegateProxy helper
- Added specification for render delegate data sources.

1.2.1 Changes:

- Fixed an error in sample code on page 7.
- Removed in-depth discussion of themes. These will be covered in a separate specification.
- Changed update() method's elem parameter name to jquery.
- Added description of getChangedDisplayProperties().
- Reworded some sentences for clarity.

Requirements

Features

Themeable Controls

SproutCore ships with many built-in controls, such as buttons, text fields, popup buttons, and so on. Often, users want to theme these controls to match the look-and-feel of their application.

Currently, theming is done solely via custom CSS. However, users are bound by the DOM structure of the built-in controls. If they need more, fewer, or different DOM elements, they must override the class and re-implement the render method. Render delegates should make it easy for views to offload their rendering to whatever their current theme is.

Component Re-use

Some views are composites of the DOM of other views. For example, SC.ListItemView can optionally display a checkbox, icon, disclosure triangle, and label. These correspond to built-in controls, such as SC.CheckboxView and SC.DisclosureView. However, for performance reasons, we want to include the DOM representations of these views without instantiating full-blown views.

Improved Performance

The SC.View API should nudge users in the direction of writing efficient custom views with fewer DOM operations.

Constraints

Backwards Compatible

Applications built using the SproutCore 1.4 gem should continue to work with the new system.

Opt-in

Users of the framework should be able to implement custom views without having to create a separate render delegate object, unless they need the features listed above.

Minimally Invasive

The implementation should requires as few modifications to the existing view layer and controls as possible, while still fulfilling all requirements.

Future-Proof

New API should be able to be adapted to up-and-coming HTML5 standards, such as SVG or canvas tags.

Specification

The Render Cycle

This section describes the render cycle as it exists today. Modifications to this implementation will be suggested in the following sections.

Creating DOM

In order to display a view in the browser, SproutCore needs to create an HTML representation that can be appended to the DOM.

To do this, SproutCore calls the view's `render` method with two parameters: an SC.RenderContext instance, and a Boolean (`firstTime`) set to `YES`, indicating that no DOM representation exists.

When `render` is called with `firstTime` set to `YES`, it is the responsibility of the view to push strings of HTML into the SC.RenderContext. Once finished, those strings will be converted into an HTML element and appended to the DOM.

Updating DOM

Certain properties of the view influence how it is displayed. For example, the appearance of SC.ButtonView will change when its `isEnabled` property changes. When one of these properties (called ***display properties***) changes, the `render` method will be called. This time, however, the `firstTime` flag will be set to `NO`. This is your cue that you should update the existing HTML representation instead of creating a new representation.

This is usually accomplished by retrieving a jQuery object for the HTML representation and making modifications to it, such as changing class names or changing the text content of the node.

Problems

Both beginner and intermediate SproutCore users often write inefficient view code for two reasons:

1. Users are unclear what the `firstTime` flag means. Often, they will ignore it and push strings into the SC.RenderContext instance every time the `render` method is called. This is grossly inefficient because it causes the old HTML representation to be thrown out and a new one to be created and replaced in DOM.

2. Even users who are familiar with the `firstTime` flag and update their HTML representation using a jQuery object will often make more modifications than necessary. Often, they will make an update for every property, rather than just the property that changed.

API Improvements

In order to clarify the difference between HTML generation and modification, the render method should be split in two.

render

The `render` method will be used exclusively for HTML generation.

Its method signature looks like this:

```
render: function(context)
```

1. *`context`* — An instance of SC.RenderContext. The strings of HTML that represent your view should be pushed into this object.

Note that the `firstTime` parameter has been dropped; the `render` method is now called only when `firstTime` would be true in the previous implementation.

update

A new method, `update`, will be called when one or more display properties are changed and an HTML representation already exists. In other words, it will be called where previously `render` would be called with `firstTime` set to `NO`.

Its method signature looks like this:

```
update: function(jquery)
```

1. *`jquery`* — A jQuery object corresponding to the HTML representation of the view.

Detecting Display Property Changes

Some DOM changes can be costly. To be efficient, the `update` method must know whether those DOM changes are even necessary. The update method may do one of two things: use SC.Object's `didChangeFor` method, or save the old value and compare.

For example, imagine you had a view with the following display properties:

```
SC.View.create({
  displayProperties: 'title value'.w()
});
```

Imagine that the view has rendered, and then one—but only one—of the display properties is updated: value.

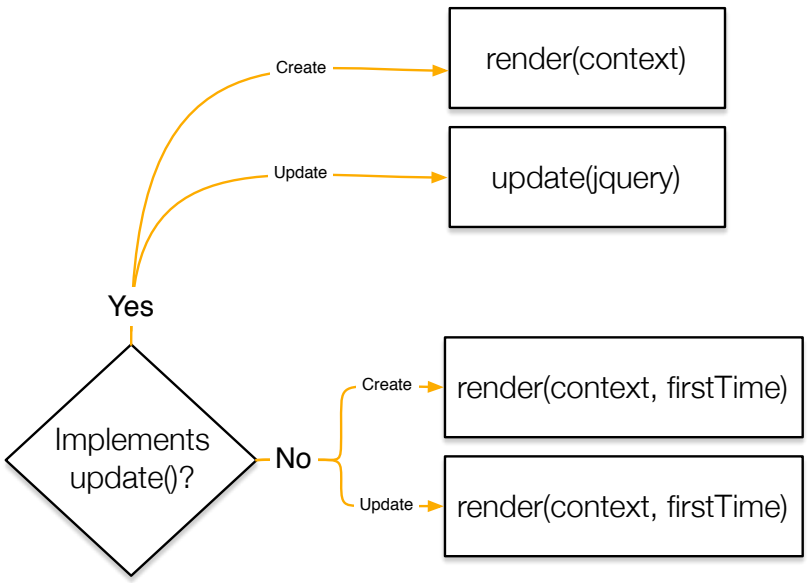
So long as `didChangeFor('update', 'title', 'value')` was called in `render` to initialize it, calling `didChangeFor('update', 'value')` during `update` will return YES. Calling `didChangeFor('update', 'title')`, conversely, will return NO.

```
render: function(context) {
  ... do rendering tasks ...

  // give didChangeFor a point of reference for the ‘update‘ method
  this.didChangeFor(‘update’, ‘title’, ‘value’);
},


update: function(jquery) {
  if (this.didChangeFor(‘update’, ‘title’)) {
    jquery.find('.title').text(this.get(‘title’));
  }

  if (this.didChangeFor(‘update’, ‘value’) {
    jquery.find('.value').text(this.get(‘value’));
  }
}
```

API Versioning

To ensure backwards compatibility, older views that only implement the `render` method (and not `update`) will continue to behave the same way.

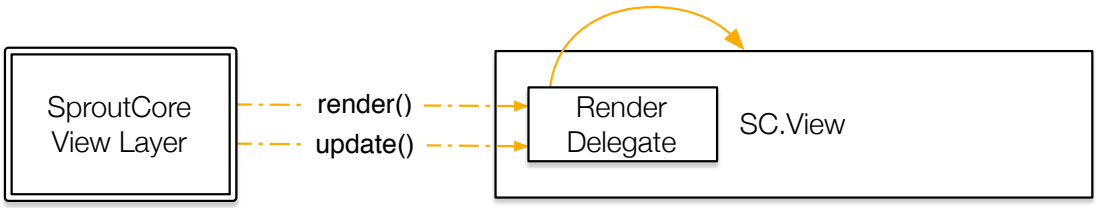
The view layer will use the existence of the `update` method to determine which version of the API should be used. For example, in a view that only has a `render` method, it will continue to get called for both creation and update, with the firstTime parameter set to the appropriate value.



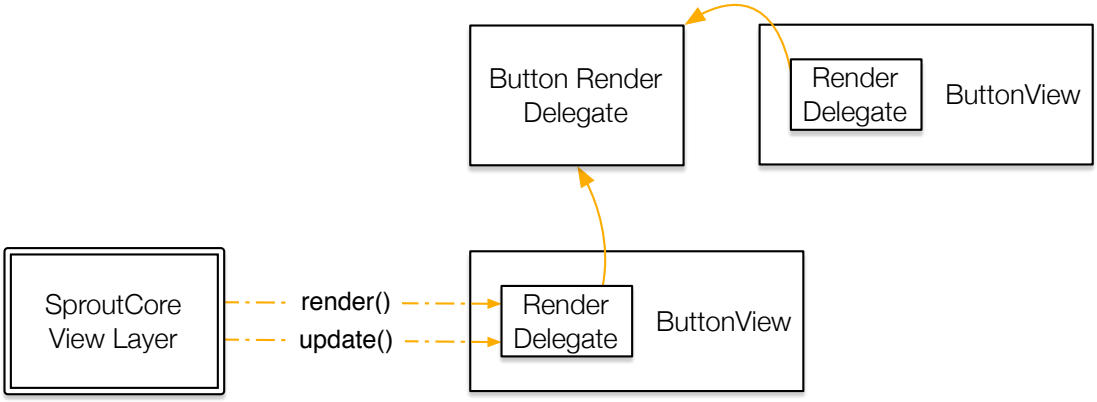
Render Delegates

Any object that implements `render` and `update` is considered to implement the ***render delegate protocol***. Any object that implements the render delegate protocol can be used as a ***render delegate***.

By default, an instance of SC.View is treated as its own render delegate. When the SproutCore view layer calls `render` or `update`, it will be called on the view because its `renderDelegate` property is set to `null`.



In some cases, you may want the logic for maintaining the HTML representation outside of the view. You can set the render delegate to be any object that implements the render delegate protocol. Because render delegates are stateless, you can share the same instance among multiple views.



If the render delegate is not a view, the first parameter passed to the `render` and `update` methods is an object representing the view being rendered. Render delegates should query this object to determine the state of the view to render.

```
render: function(dataSource, context)
```

1. *`dataSource`* — The object that contains the display properties that determine how and what it should be rendered.

2. *`context`* — An instance of SC.RenderContext. The strings of HTML that represent your view should be pushed into this object.

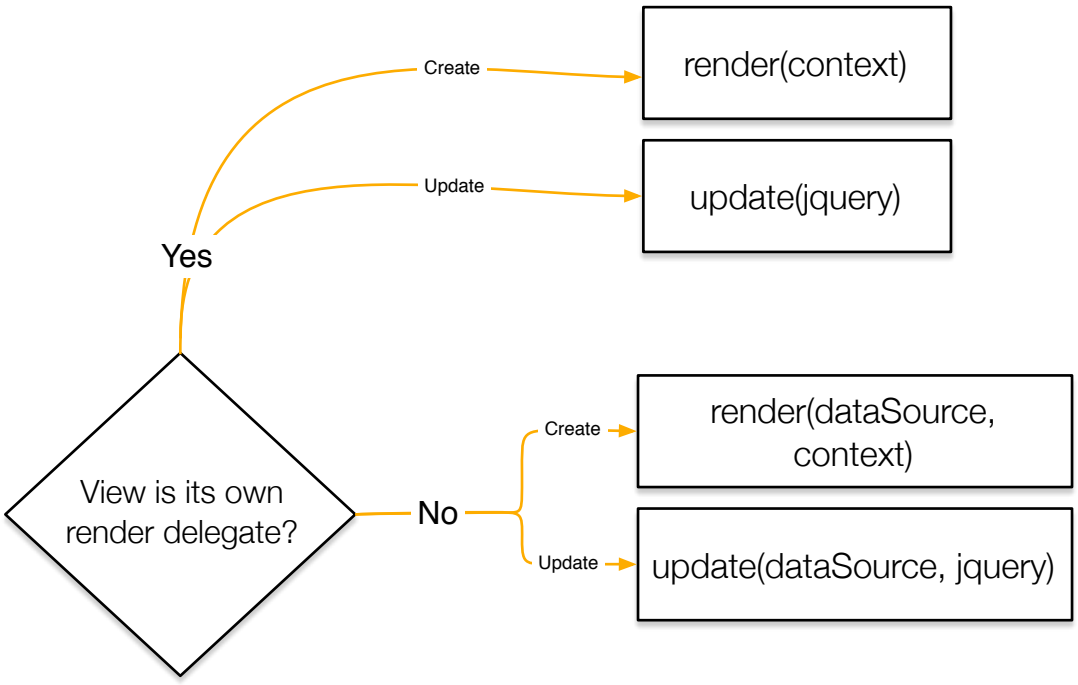
```
update: function(dataSource, jquery)
```

1. *dataSource* — The object that contains the display properties that determine how and what it should be rendered.

2. *jquery* — A jQuery object corresponding to the HTML representation of the view.

For example:

```
render: function(dataSource, context) {
  context.setClass('sel', dataSource.get('isSelected'));
}
```



SC.RenderDelegate

SproutCore offers an *optional* base class for render delegates. Helper methods can be mixed in to this base class, and then be available to all render delegates.

For example, the includeSlices method allows you to easily add a bunch of DOM elements that work automatically with Chance's `@include slices()` directive to give your controls multi-slice scalable images.

For example:

```
// add a helper
SC.mixin(SC.RenderDelegate.prototype, {
  myHelper: function(dataSource, context, ...) {
    // do some helping...
    context.addClass('blah');

    // if any of it should be specialized by theme, you can
    // call a render delegate here to delegate it:
    dataSource.get('theme').otherRenderDelegate.render(...);
  }
});


// use the helper
MyTheme.myRenderDelegate = SC.RenderDelegate.create({
  render: function(dataSource, context) {
    this.myHelper(dataSource, context, someOtherArg);
  }
});
```

These helpers are meant to be general and non-theme-specific. If a helper should take action that *is* theme-specific, consider delegating it with a render delegate.

Data Sources

Data sources implement a subset of `SC.Object`'s API, along with a few special required properties.

The data source API must be simple so that anyone—not just SC.Views—can create them. As such, it would be best if it did not gain many more required properties or methods. However, it could gain additional *optional* API; for instance, it could add optional methods to set up animation timers.

Methods

- `get('propertyName')`
- `didChangeFor('propertyName')`

Properties

- `theme:` The theme being used for rendering.
- `renderState:` An empty hash for use by the render delegate. While render delegates are usually completely stateless, sometimes they need to store some sort of state.

Render Delegates & Views

SC.View does not pass *itself* to its render delegate as the data source.

As both a limited form of protection and a convenience to avoid API collision, views pass *proxies* to their render delegates. These proxies implement the dataSource protocol, and have two basic tasks:

1. **Limit to displayProperties.** Calling `.get()` on the proxy will only return properties that are in the view's `displayProperties` array. The view's displayProperties are the properties it provides to match its render delegate's API, so only those properties should be passed.

2. **.get('propertyName') -> .get('displayPropertyName').** When `.get('aProperty')` is called on the proxy, it will first check to see if `'displayAProperty'` is defined in `displayProperties.` If it is, `displayAProperty` will be used instead of `aProperty`.

   This allows you to define `displayTitle` and have the render delegate automatically use it instead of `title.` This is used by SC.SliderView: it has a displayValue property that converts its 'value' property from a number between its '`minimum`' and '`maximum`' to a number between 0 and 1; when the `sliderRenderDelegate` asks for '`value`', the proxy provides it with '`displayValue`' from the view.

   Note: '`displayValue`', not '`value`', is the property that should be listed in the displayProperties array.

Render Delegates & Themes

Controls can take on a different appearance depending on their context. For example, in Mac OS X, consider these four variations of NSButton:

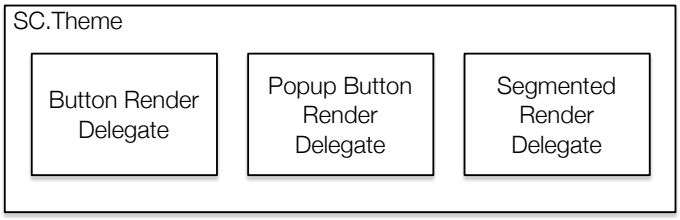
They are all instances of NSButton and exhibit identical behavior. The only difference is their visual appearance.

In SproutCore, a control's theme dictates how it is rendered. Themeable controls retrieve their respective render delegate from an instance of SC.Theme and let it be responsible for creating the HTML representation.

To learn more about themes, see the specification at http://www.sproutcore.com/specifications/.

Registering a Render Delegate

The collection of render delegates that compose a unified appearance for controls is contained in an instance of SC.Theme.



Adding a render delegate to a theme is as simple as adding a property to it:

```
MyCustomTheme.buttonRenderDelegate = SC.RenderDelegate.create({
  render: function(dataSource, context) {
    // Your render code here.
  },

  update: function(dataSource, context) {
    // Your update code here.
  }
});
```

Building Themeable Controls

You can tell a view to retrieve a render delegate from the theme by setting its `renderDelegateName` property. The view will use this property to find the associated render delegate from the current SC.Theme object.

For example, SC.ButtonView has its `renderDelegateName` property set to `"buttonRenderDelegate"`. When a new instance of SC.ButtonView is created, it looks up the `buttonRenderDelegate` property on its theme and saves that object to the `renderDelegate` property.